CS2100 Cheatsheet

github.com/reidenong/cheatsheets, AY23/24 S2

Number Systems

Byte : 8 bits, Word : n bytes, `char` : 1 byte
`int` , `float` : 4 bytes, `double` : 8 bytes

Negative Powers of 2

power	value			
2^{-1}	0.5		2^{-2}	0.25
2^{-3}	0.125		2^{-4}	0.0625
2^{-5}	0.03125		2^{-6}	0.015625
2^{-7}	0.0078125		2^{-8}	0.00390625

Unsigned Numbers

ASCII: 7 bits, 1 parity bit at MSB
* Even Parity: Total number of 1s is even

Excess-x Representation
Smallest Value: 00...000 = $-x$;
Largest Value: 11...111 = $x - 1$;
Zero: $(x)_N$

Signed Numbers

Sign-and-Magnitude
1 sign bit, $N - 1$ magnitude bits.
Largest value: 0111...111 = $2^{N-1} - 1$
Smallest value: 1111...111 = $-(2^{N-1} - 1)$
Zeros: 0000...000 or 1000...000

1s Complement
$-x = 2^N - x - 1$
Largest value: 0111...111 = $2^{N-1} - 1$
Smallest value: 1000...000 = $-2^{N-1} + 1$
Zeros: 0000...000 or 1111...111

Negation: Flip all bits
+/-: If there is a carry out of MSB, add 1
Overflow: Result is opposite sign of A and B

2s Complement
$-x = 2^N - x$
Largest value: 0111...111 = $2^{N-1} - 1$
Smallest value: 1000...000 = -2^{N-1}
Zeros: 0000...000

Negation: Flip all bits and add 1
+/-: Ignore the carry out
Overflow: Result is opposite sign of A and B

Fixed Point Representation

Number of bits allocated to whole number and fraction are fixed.

IEEE 754 Floating Point Representation

Single Precision (32 bits)
1-bit sign, 8-bit exponent (excess-127), 23-bit mantissa

Double Precision (64 bits)
1-bit sign, 11-bit exponent (excess-1023), 52-bit mantissa

* If exponent is all 1s, ⇒ x = NaN
If exponent is all 0s, ⇒ x = 0.

EG. $-6.5_{10} = -110.1_2 = -1.101_2 \times 2^2$

1	10000001	101000000....000
sign	exponent (excess-127) 0 = 01_111_111	mantissa **(normalised to 1)**

C Programming

Format Specifiers

%c	char	printf/scanf
%d	int	printf/scanf
%f	float/double	printf
%f	float	scanf
%lf	double	scanf
%e	float/double	printf (for scientific notation)
%p	addresses	printf (in hexadecimal)
prefix 0	octal	
prefix 0x	hexadecimal	

Strings

Input/Output

Format Specifiers

\n	New line
\t	Horizontal tab
\"	Double quote
%%	Percentage sign
\	Backslash
\0	Null character

Initializing, reading and writing:

```
char str[] = "apple";
char str[] = {'a', 'p', 'p', 'l', 'e', '\0'};
fgets(str, size, stdin);
// reads size-1 chars, or until newline
scanf("%s", str); // reads until whitespace
puts(str);  // terminates with newline
printf("%s", str);
```

`fgets` also reads in the newline if present, and may need to be rectified with
`if (str[len - 1] == '\n') str[len - 1] = '\0';`

`char` is 1 byte
Other Functions:
`strlen(str)` returns length of string
`strcpy(dest, src)` copies src to dest
`strcat(dest, src)` appends src to dest
`strcmp(str1, str2)` compares str1 and str2, returns 0 if equal, −1 if str1 < str2, 1 if str1 > str2

Pointers and structs

for `struct object_t` , to refer to a pointer to variable `a` , we can use
1. `object_t *object_p = &object1;`
`(*object_p).a = 1;`
2. `object_p->a = 1;`

Pass by Value / Reference

A function can only modify an object if the argument is directly a pointer to the object or an array (not array object).

MIPS

Major Components in a computer:

- Processor: performs computations
- Memory: stores data and instructions
- Bus: connects the processor and memory

Registers:

- Limited in number (16-32)
- Faster than memory
- No data type, instruction assumes data stored is of the correct type
- MIPS has 32 Registers:

Name	Number	Use
`$zero`	0	Constant 0, cannot write to this
`$at`	1	Reserved for assembler
`$v0-$v1`	2-3	Values for results and expression evaluation
`$a0-$a3`	4-7	Arguments to functions
`$t0-$t7`	8-15	Temporary data
`$s0-$s7`	16-23	Saved data
`$t8-$t9`	24-25	Temporary data
`$k0-$k1`	26-27	Reserved for OS kernel
`$gp`	28	Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

Arithmetic Instructions

- `NOT` can be achieved by `NOR $t0, $t0, $zero` or `XOR $t0, $t0, 0b111...111`
- Loading a 32 bit constant into a register can be done by `LUI $t0, 0xAAAA` (set the upper 16-bits) and `ORI $t0, $t0, 0xF0F0` (set the lower-order bits).
- `C16` are not sign-extended as it is treated as raw bits and not a number, but `C16_2s` is sign-extended.

MIPS Memory instructions:

- 32 registers, each 32 bit Loading
- Each word contains 32 bits
- Memory addresses are 32 bit Long, with 2^{30} memory addresses available
- Registers do not have data types, and is interpreted according to the instruction:
- load/store operations `lw $t0, 4($t1)` means $t0 contains a memory address, `sw $t0, 4($t1)` means $t0 contains a memory address
- `lhw` load half word is a pseudo instruction that does `lw` and shifts the result right by 16 bits
- offset must be a multiple of 4 for memory access

Accessing arrays

```
# Accessing arrays by index
  addi $s0, $zero, 0          # idx = 0
loop:
  add  $t0, $s0, $s4           # $t0 =idx+add
  lb   $t1, 0($t0)             # $t1 = A[idx]
  addi $s0, $s0, 1             # idx++
  bne  $s0, $s5, loop          # idx!=n, loop
# Accessing arrays by pointer
  add  $t0, $s4, $s0         # add of A[*s0]
  add  $t1, $s4, $s1         # add of A[end]
  addi $t1, $t1, -1          # add of
A[end-1]
  loop:
    lb   $t2, 0($t0)         # $t2 = *$t0
    addi $t0, $t0, 1         # $t0++
    bne  $t0, $t1, loop      # $t0!=end, loop
```

MIPS Decision Making

Instruction	Operation
`beq $t1, $t2, L1`	Branch if equal
`bne $t1, $t2, L1`	Branch if not equal
`j L1`	Jump to L1
`slt $t0, $t1, $t2`	Set $t0 = 1 if $t1 < $t2, else 0
`slti $t0, $t1, C`	Set $t0 = 1 if $t1 < C, else 0
`bgt $t1, $t2, L1`	Branch to L1 if $t1 > $t2 (Pseudo)
`blt $t1, $t2, L1`	Branch to L1 if $t1 < $t2 (Pseudo)
`bge $t1, $t2, L1`	Branch to L1 if $t1 >= $t2 (Pseudo)
`ble $t1, $t2, L1`	Branch to L1 if $t1 <= $t2 (Pseudo)

PC-Relative Addressing

- Memory address is 32 bits, but `immediate` is only 16 bits -> There is a need to use relative addressing
- range of branch is $\pm 2^{15}$ bytes from the PC (one bit is used for sign)
- if branch is taken, **PC = PC + 4 + (immediate × 4)**
- if branch is not taken, **PC = PC + 4**

J-format (Jump format)

`opcode`	`target address`
6 bits	26 bits

- Jumps are word-aligned, so last 2 bits are always 00
- Target address is shifted right by 2 bits and concatenated with the upper 4 bits of the PC+4
- range of jump is 256 MB (2^{28} bytes)
- Due to using the first 4-bits, we can only jump within our block

MIPS Functions

Pseudo Load instructions

- `la $rd, add` loads the address into the register
- `li $rd, imm` loads the immediate value into the register

```
# la $t0, address
lui $t0, 0xAAAA
# (upper 16 bits of address)
ori $t0, $t0, 0xBBBB
# (lower 16 bits of address)
# li $t0, 0xAAAABBBB
lui $t0, 0xAAAA
ori $t0, $t0, 0xBBBB
```

Jumping functions

- `jal L1` jumps to L1 and stores the return address in `$ra`
- `jr $t0` jumps to the address in `$t0`

Function calls

We load the system call code into register `$v0` and the arguments into `$a0` - `$a3`, then call `syscall`. Return values are stored in `$v0`.

Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_character	11	$a0 = character	
read_character	12		character (in $v0)
open	13	$a0 = filename,	file descriptor (in $v0)
		$a1 = flags, $a2 = mode	
read	14	$a0 = file descriptor,	bytes read (in $v0)
		$a1 = buffer, $a2 = count	
write	15	$a0 = file descriptor,	bytes written (in $v0)
		$a1 = buffer, $a2 = count	
close	16	$a0 = file descriptor	0 (in $v0)
exit2	17	$a0 = value	

```
# Printing an integer
  li $v0, 1
  li $a0, 10
  syscall
```

MIPS Datapath

Instruction Execution cycle

1. Fetch: Read the instruction from memory
2. Decode: Determine the operation to be performed
3. Operand Fetch: Get required operands
4. Execute: Perform the operation
5. Result Write: Store the result

1. Fetch

- Use PC to get instruction from memory
- PC is incremented by 4 to get the next instruction
- **output**: instruction to be decoded

2. Decode

- Read opcode to determine instruction
- Read registers to get operands
- **output**: operations and operands

3. ALU (Arithmetic Logic Unit)

- Perform arithmetic and logical operations

> Arithmetic (`add` , `sub`)
> Shifting (`sll`)
> Logical (`and` , `or`)
> Memory address calculation (`lw` , `sw`)
> Branch register comparison and target address calculation

- **output**: calculation result

When the ALU is handling branch instructions, there are 2 considerations.

1. Branch Outcome: Use ALU to compare the registers, determine if the branch is taken.
2. Target Address: Use ALU to calculate the target address, requires PC from Fetch stage and Offset from decode stage.

4. Memory

- Only `load` and `store` instructions need this stage with memory address from ALU

5. Register Write

- Writes the result back into register
- Write Data was generated in the decode stage

MIPS Control

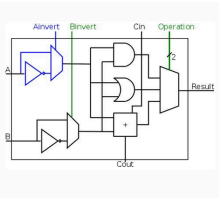
Control Signals

- Control signals are generate based on the opcode and funct fields

`RegDst`	• Decode/Operand fetch stage • Selects the destination register (0) : Write register = inst[10:16] (1) : Write register = inst[15:11]
`RegWrite`	• Decode/Operand fetch stage • Write to register (0) : No write to register (1) : Write to register
`ALUSrc`	• ALU stage • Selects the second ALU operand (0) : Second operand = register value (1) : Second operand = sign-extended immediate value (inst[15:0])
`ALUcontrol`	• ALU stage • Select operation to be performed
`MemRead` / `MemWrite`	• Memory stage • Read/Write from memory (0) : No read/write from memory (1) : Read/write from memory
`MemtoReg`	• RegWrite stage • Selects the memory result to write to register (0) : Write register = ALU result (1) : Write register = Memory result
`PSCrc`	• Memory/RegWrite stage • Select the next PC value • PCSrc = Branch **AND** isZero (0) : PC = PC + 4 (1) : PC = PC + 4 + (sign-extended immediate value * 4)
`ALUop`	• Generated in the decode stage using opcode, and is sent to the ALU Control unit to generate ALUcontrol.

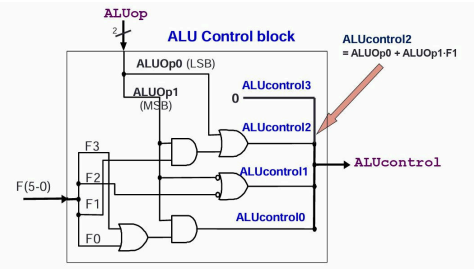
1-bit ALU

ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR



ALU Control Unit

The ALU Control block receives input from the opcode in the form of `ALUop`, and the funct field, to generate the ALUcontrol signal.



MIPS Control Unit

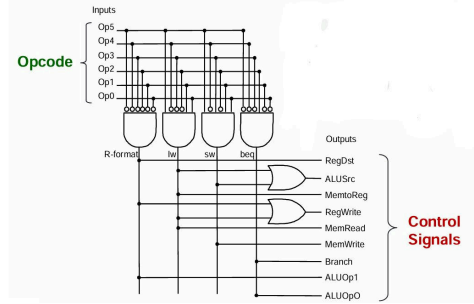
Inputs:

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

Outputs:

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	0	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Circuit:



Critical Paths

R-type / imm instructions

Inst Mem → RegFile → MUX (ALUsrc) → ALU → MUX (MemtoReg) → RegFile

`lw` / `sw` instructions

Inst Mem → RegFile → ALU → DataMem → MUX (MemtoReg) → RegFile

`beq` instructions

Inst Mem → RegFile → MUX (ALUsrc) → ALU → AND → MUX (PCSrc)

Overall Cycle Time:
1. Find the longest latency, t.
2. Take the lower bound of $\frac{1}{t}$.

Instruction Execution
1. Read contents of storage elements (Registers, Memory)
2. Perform computations
3. Write results to a storage element

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

All these have to be performed within a single clock cycle.
Solutions:
1. Multicycle Implementation
- Break each instruction into execution steps, where each step takes one clock cycle
- Instructions take variable number of cycles to execute

2. Pipelining
- Allows different instructions to be executed in parallel

Boolean Algebra

Consensus Theorem

$$A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$$
$$(A + B) \cdot (A' + C) \cdot (B + C) = (A + B) \cdot (A' + C)$$

A minterm of n variables is a product of n literals, whereas a maxterm of n variables is a sum of n literals.

$$M(3) = M(0011) = A + B + C' + D'$$
$$m(3) = m(0011) = A' \cdot B' \cdot C \cdot D$$
$$M(3) = m(3)'$$

Each minterm is the complement of the maxterm, and vice versa.

$\forall \alpha, \beta \in \mathbb{R}$, if $\alpha \neq \beta$, we have

$$m\alpha \cdot m.\beta = 0$$
$$M\alpha + M\beta = 1$$

Logic Gates

AND		
A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

NAND		
A	B	(A · B)'
0	0	1
0	1	1
1	0	1
1	1	0

XOR		
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR		
A	B	(A ⊕ B)'
0	0	1
0	1	0
1	0	0
1	1	1

NOR		
A	B	(A + B)'
0	0	1
0	1	0
1	0	0
1	1	0

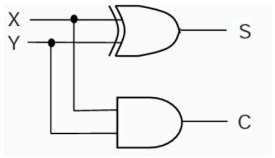
{AND, NOT}, {OR, NOT}, {NAND}, {NOR} are all complete sets.

An SOP expression can be implemented by a
- 2 level AND-OR Circuit
- 2 level NAND Circuit

An POS expression can be implemented by a
- 2 level OR-AND Circuit
- 2 level NOR Circuit

Gray codes
- Adjacent codes differ by only 1 bit
- n bits $\rightarrow 2^n$ codes
- Generated by flipping outermost bits of previous codes

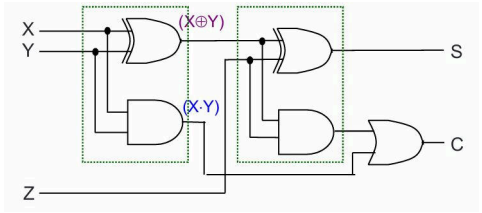
Half Adder



Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$C = X \cdot Y$$
$$S = X' \cdot Y + X \cdot Y' = X \oplus Y$$

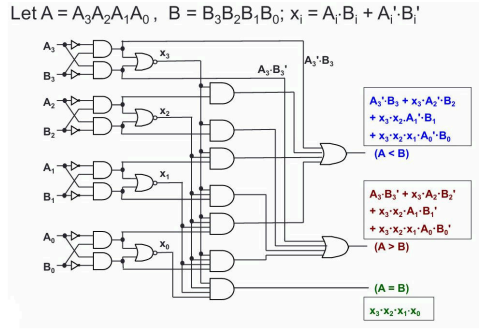
Full Adder



$$C = X \cdot Y + X \cdot Z + Y \cdot Z$$
$$= X \cdot Y + (X \oplus Y) \cdot Z$$

$$S = X \oplus Y \oplus Z$$
$$= X' \cdot Y \cdot Z + X \cdot Y' \cdot Z + X \cdot Y \cdot Z' + X \cdot Y \cdot Z$$

Magnitude Comparator



Circuit Delays
- n-bit ripple-carry parallel adder will have delays of

$$S_n = (2(n-1) + 2)t$$
$$C_{n+1} = (2(n-1) + 3)t$$
$$\text{Max Delay} = C_{n+1}$$

Where t is the delay of a single full adder.

MSI Components

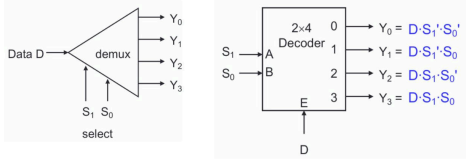
Decoders
- Converts n input lines to 2^n output lines
- Each output line is a minterm of the input lines

Encoders
- Converts 2^n input lines to n output lines
- Each output line is a Sum of the input bits
- Priority Encoder: Outputs the highest order bit that is 1

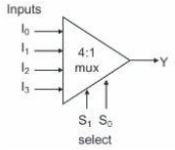
Example of a 4-to-2 priority encoder:

Inputs				Outputs		
D_0	D_1	D_2	D_3	f	g	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Demultiplexer
- Chooses which output a input line goes to based on selection lines
- 2^n outputs, n selection lines



Multiplexer
- Chooses which input line goes to the output line based on selection lines
- Can be implemented using a decoder and a OR gate

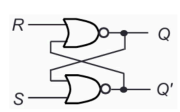


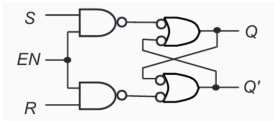
I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

$$Y = I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3$$

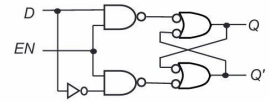
Sequential Circuits
- Latches are pulse triggered
- Flip-flops are edge-triggered
- Circuit is self correcting if any unused state can transit to a used state after a finite number of cycles

S-R Latch (+ Gated)





Gated D Latch



S-R Flip-flop
D Flip-flop
J-K Flip-flop

$$Q^+ = J \cdot Q' + K' \cdot Q$$

T Flip-flop

$$Q^+ = T \oplus Q = T \cdot Q' + T' \cdot Q$$

Excitation Tables

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

SR Flip-flop

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

D Flip-flop

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop

Pipelining
1. **IF**: Instruction Fetch
2. **ID**: Instruction Decode & Register Read
3. **EX**: Execute an operation or calculate an address
4. **MEM**: Access memory
5. **WB**: Write back the result

Register	Stored information
IF/ID	• Register numbers for reading 2 Reg • 16-bit offset to be sign extended to 32-bit • PC + 4
ID/EX	• Data values read from register file • 32-bit immediate value • PC + 4

EX/MEM	• (PC + 4) + (immediate x 4) • ALU result • isZero? signal • Data Read from register file
MEM/WB	• ALU result • Memory Read Data

- Write Register number follows the instruction through the pipeline until it is needed in WB stage

Pipelining Performance

Single-Cycle Processor
- Cycle Time, CT_{seq} = max(Time needed for a **instruction**)
- Execution Time for I instructions = $I * CT_{seq}$

Multi-Cycle Processor
- Cycle Time, CT_{multi} = max(Time needed for a **stage**)
- Execution Time for I instructions
  = $I * CT_{multi}$ * (Average CPI)

Pipelining Processor
- Cycle Time, CT_{pipe}
  = max(Time needed for a **stage**) + (Pipelining Overhead)
- # of Cycles for I instructions
  = $I + N - 1$ ($N - 1$ cycles to fill pipeline)
- Execution Time for I instructions
  = $(I + N - 1) * CT_{pipe}$

Pipeline processor can gain N times speedup, where N is the number of stages in the pipeline.

$$\text{Speed up} = \frac{\text{Time}_{seq}}{\text{Time}_{pipe}}$$
$$= \frac{I \times N \times T_1}{(1 + N - 1) \times T_1}$$
$$\approx \frac{I \times N \times T_1}{I \times T_1} = N$$

Data Dependencies
- **Read After Write (RAW)**: Data hazard where the data is read before it is written
- **Write After Read (WAR)**: Data hazard where the data is written before it is read
- **Write After Write (WAW)**: Data hazard where the data is written before it is written

Solutions
- **Forwarding**: Send the data directly to the next stage
- **Stalling**: Insert a bubble into the pipeline

Control Dependency

To avoid the 3-cycle stall for branch instructions, there are 3 solutions:

1. Early Branch Resolution
- Resolve the branch in the ID stage (smaller stall may be needed)

2. Branch Prediction
- Assume all branches are not taken
- If assumption is wrong, flush pipeline

3. Delayed Branch
- There is an instruction preceding the branch which can be moved into the delayed slot

Types of forwarding Paths:

Control Dependency for early branching
- ALU to ID
- MEM to ID

RAW hazard
- ALU to ALU
- MEM to ALU
- ALU/MEM to MEM (xxx → `sw`)

Mem-to-Mem (`lw` → `sw`)
- MEM to MEM

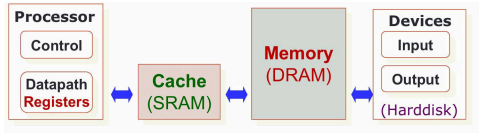
Stall finding
- RAW Data hazards
- All branches / loops

Cache

SRAM
- Low Density (6 transistors per memory cell)
- Fast access latency 0.5 - 5 ns
- Flip flops

DRAM
- High density (1 transistor per memory cell)
- Slow access latency 50 - 70 ns
- Main memory

Memory Hierarchy
- Small but fast memory near CPU
- Large but slow memory further away from CPU
- Registers → SRAM → DRAM → Disk



Principle of locality states that the program accesses onyl a small portion of the memory address space within a small time interval
- Temporal locality: If a item is referenced, it will be referenced again soon
- Spatial locality: If a item is referenced, nearby items will tend to be referenced soon

Memory Access Time
- **Hit**: Data is in cache
  - **Hit rate**: Fraction of memory accesses that are hits
  - **Hit time**: Time to access data in cache
- **Miss**: Data is not in cache
  - **Miss rate**: Fraction of memory accesses that are misses (1 − Hit rate)
  - **Miss penalty**: Time to replace cache block + Hit time

Average Access Time
= Hit rate × Hit time + Miss rate × Miss penalty

As block size increases,
+ Spatial Locality improves → Hit rate improves
- Longer time to fill block → Miss penalty increases
- Block size too big → Too few cache blocks → Miss rate increases

Cache Write Policy
- **Write-through**: Write to cache and memory, use a write buffer so write to memory can be done in parallel
- **Write-back**: Write to cache only, write to memory when block is replaced. Need a dirty bit to indicate if block is modified

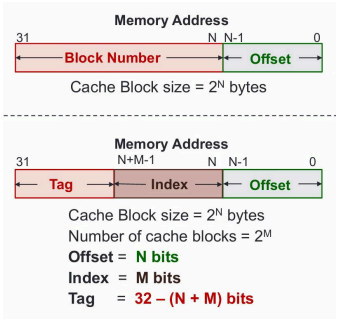
Miss types
- Compulsory (Cold): First access to a block
- Conflict: Several blocks are mapped to the same block/set
  - Direct Mapped / Set Associative Cache
- Capacity: Cache is too small to hold all blocks, blocks are discarded
  - Fully Associative Cache

Handling Misses
- Read miss: Load block from memory to cache
- Write miss:
  - Write allocate: Load block from memory to cache, write to cache. Continue with current write policy
  - Write around: Write directly to memory

Direct Mapped Cache

Contents:
- Data Block (line)
- Tag of the memory block + valid bit + dirty bit

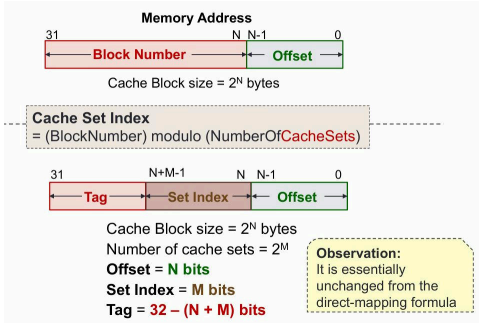
Cache Hit:
(Valid[index] = 1) AND (Tag[index] = Tag[Memory Address])



N-way Set Associative Cache

Helps to tackle the problem of conflict misses.

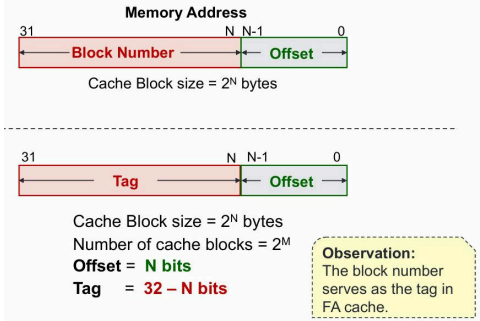
Cache has sets, which each contain N blocks. Within the set, a memory block can be placed in any of the N blocks.



A direct mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2.

Fully Associative Cache
- Each block can be placed in any cache block

+ Can be placed in any location
+ No conflict misses
- Need to search all cache blocks



$$\text{Total Miss} = \text{Compulsory Miss} + \text{Conflict Miss} + \text{Capacity Miss}$$
$$\text{Capacity Miss(FA)} = \text{Total Miss(FA)} - \text{Compulsory Miss(FA)}$$

SA / FA Block Replacement policy
- **Least Recently Used (LRU): Replace the block that has not been used for the longest time**
  - Temporal Locality
- First In First Out (FIFO): Replace the block that has been in the cache the longest
- Random: Replace a random block
- Least Frequently Used (LFU): Replace the block that has been used the least

Improving Miss penalty
- Separate Data and instruction cache
- Unified Cache

Summary

Direct Mapped	N-way SA	FA
Block can only be placed in one location defined by index	Block can be placed in any one of N blocks in the set defined by index	Block can be placed in any cache block
Match tag in the only location the block can be in	Parallel match tags for all blocks in set	Parallel match tags for all blocks in cache
Only one block can be replaced	Replacement policy	Replacement policy

Write Strategies
- Write policy: Write-through / Write-back
- Write Miss policy: Write allocate / Write around

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

! ! ! For MIPS, char = 1 byte, use `lb` and `sb`